Progress-Sensitive Security for SPARK

Willard Rafnsson[1], Deepak Garg[2], and Andrei Sabelfeld[3]

[1]Carnegie Mellon University, USA
[2]Max Planck Institute for Software Systems, Germany
[3]Chalmers University of Technology, Sweden

Abstract. SPARK 2014 is a safety critical language subset of Ada developed by Altran and used for developing safe and secure software by major industrial players in the aviation, commercial, medical, space, and military domains. This paper puts a spotlight on the SPARK flow analysis. Articulating the boundaries of what is achievable by the analysis, we spell out attacks to exploit termination, progress, resource exhaustion, and timing channels. We harden the analysis to achieve security against stronger attackers, with the focus on progress-sensitive security as our baseline. Instead of redesigning and reimplementing the enforcement, we leverage known flow analyses for weaker attackers by a transform on program dependence graphs. We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. A case study, derived from publicly available code for a control unit of a missile, indicates the usefulness of the approach.

1 Introduction

SPARK is a safety critical language subset of Ada developed by Altran and used by industry in the aviation, commercial, medical, space, and military domains. Applications range from programming jet engines (Lockheed Martin) to military aviation (EuroFighter), UK's air traffic control system (Altran), cross-domain guards (Rockwell Collins), smart card OS (MULTOS), biometrics software (NSA), and multi-level security systems (Secunet) [42].

SPARK 2014. A recent major overhaul of SPARK has led to SPARK 2014 [44], a language and accompanying tools for developing safe and secure software. To aid security verification, a flow analysis is integrated in the compiler to track information flow in SPARK programs and is used in applications like separation kernels [29] and multi-level workstations [39].

Information flow security. The security model of SPARK programs draws on information flow tracking. The goal is to track the propagation of data from sources (inputs) to sinks (output) as information is manipulated by programs. For systems whose sources and sinks are classified into secret and public (or more complex classifications [18]), the baseline policy is *noninterference* [16,21] that prevents secret inputs from affecting public outputs.

There are different ways in which noninterference can be broken, corresponding to different information flow channels. An *explicit* flow results from a data

flow from the right-hand side to the left-hand side in an assignment. An *implicit* [19] flow is via control flow: for example, branching on a secret and outputting different public values in the branches is an implicit flow that leaks information about the secret without any explicit leaks. The *termination channel* [48] is another source of potential leaks: a program that loops on a secret and outputs a public value on exiting the loop reveals whether the loop has terminated and therefore leaks information about the secret guard. A generalization of this channel is the *progress channel* [7] that can be used to leak information about secrets via the progress of public outputs. In contrast to the one-bit termination channel, this channel allows leaking secrets in their entirety by brute force attacks [7]. Other channels of interest are *resource exhaustion* and *timing* [37], which allow the attacker to learn secret information by observing abnormal behavior and time variation, respectively.

SPARK security examined. Usage of the SPARK flow analysis in industry is encouraging. It makes the following questions important. What attacks does it prevent? How can it be extended to achieve security against more powerful attackers? Can it lead to a general methodology applicable to similar analyses?

This paper puts a spotlight on the flow analysis in SPARK GPL 2015. Released April 28 2015, it is, as of January 1 2016, the latest GPL edition of SPARK 2014. To articulate the boundaries of what it can achieve, we demonstrate that the analysis successfully tracks explicit and implicit flows and spell out attacks to exploit termination, progress, resource exhaustion, and timing channels.

SPARK security improved. With the goal to harden the analysis against stronger attackers, we set our baseline at the progress-sensitive security policy [31,8,12,32]. This policy is a natural generalization of noninterference to programs with output, in contrast to its progress-insensitive counterpart [7,8,12] that needs to carve out leaks due to computation progress. Further, as mentioned earlier, ignoring the progress channel implies opening up brute force leaks that may extract secrets in their entirety. Our key goal is to design a general approach that allows leveraging existing analysis and tools for explicit and implicit flows, such as SPARK flow analysis to enforce the stronger progress-sensitive security. This goal is particularly important given the state of the art where the vast majority of the information flow tools in addition to SPARK (e.g. FlowFox [22], JSFlow [24] and IFC4BC [10] for JavaScript, Jif [30], Paragon [14] and JOANA [23] for Java, FlowCaml [40] for Caml, all discussed in Section 8) are currently only able to enforce progress-insensitive security.

Achieving progress-sensitive security. With this main goal at hand, the core idea for enforcement is as follows. We set out to leverage two independent components: graph-based analysis for explicit/implicit flows and termination analysis. There have been many successful efforts on developing such components, with the above-mentioned information flow tools for the former and much encouraging progress on the latter [17,27,45]. Facilitated by the latter, Moore et al. [28] show how to use *termination oracles* for termination-sensitive information flow analysis. Similarly, we parametrize our approach in the termination analysis to determine which loops terminate and perform a graph transform on the program

dependence graph where we represent termination and progress flows by injecting additional edges going out of potentially diverging loops. This lets us reuse graph-based analyses, e.g. the one by Horwitz [26] that is behind the SPARK flow analysis, since we can simply apply it to the transformed graph. The elegance of this approach is that even if a trivial termination analysis ("all loops might diverge") is plugged into the framework, we get a sound and meaningful enforcement of progress-sensitive noninterference corresponding to Smith's and Boudol and Castellani's canonical restrictions for the termination channel [41,13].

We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. We apply the source-to-source transformation on a case study with a control unit of a missile, loosely based on publicly available code by Hilton [25]. We formulate desired properties, such as "the orientation sensors may not affect self-destruction", in terms of information-flow policies and demonstrate how our enforcement verifies these properties.

Contributions. The paper's major contributions are (i) the attacks on the SPARK flow analysis to demarcate its boundaries, (ii) leveraging a progress-insensitive SPARK flow analysis (by changing the analyzer conservatively, or through source-to-source transformation) to enforce progress-sensitive noninterference, and (iii) a case study with a missile code controller to demonstrate the usefulness of the approach. While our work is motivated by improving the SPARK flow analysis, we believe the overall idea is portable to other approaches and tools. Thus, we present our results more generally. For example, our framework is graph-based, which opens up possibilities for natural adoption to other graph-based tools such as JOANA [23]. Combining the major and minor contributions, the paper contributes the following:

- Attacks illustrating the boundary of what SPARK's flow analysis can achieve, leaking via termination, progress, resource exhaustion, and timing (Section 2);
- A policy framework for expressing progress-(in)sensitive security conditions (Section 4) for an imperative language (Section 3) at the heart of SPARK;
- A general graph-based approach for dependency analysis using termination oracles to achieve progress-sensitive security (Section 5);
- A general graph-based framework for dependency analysis of reactive programs, also distinguishing output content from output presence (Section 5);
- Soundness of the graph-based enforcement for the core language (Section 5);
- Source-to-source transform leveraging existing graph-based flow analyses in a modular fashion (Section 6) to achieve progress-sensitive security; and
- Case study with a control unit of a missile that verifies desired security properties (Section 7).

Our code compiles with GNAT GPL 2015. Released April 28 2015, it is, as of January 1 2015, the latest GPL edition of the Ada 2012 compiler. Our code can be found online [49].

Scope. While resource exhaustion and timing channels are important, we leave their consideration and exploration of more sophisticated attacks on the SPARK

```
procedure Leak (H : in out Byte) is
begin
  H := H;

-- if H is even: nontermination.
-- else terminate with output "!".
  if H mod 2 = 0 then
    while True loop
      H := H;
    end loop;
  end if;
  Write (Standard_Output,
         Character'Pos('!'));
end Leak;
```

```
procedure Leak (H : in out Byte) is
  K : Byte := 0;
begin
  H := H;
  while True loop
    Write (Standard_Output, K);
    if K >= H then
      while True loop
        H := H;
      end loop;
    end if;
    K := K + 1;
  end loop;
end Leak;
```

Fig. 1: Termination leaks (left) and progress leaks (right) in SPARK

security analysis for future work. Typically, attacks on these channels require more efforts from the attacker and result in attacks with lower bandwidth [37]. For similar reasons, we leave declassification [38] out of the scope of the present work. Although important and wished for by the SPARK developers [35], the flow analysis in SPARK is useful even without declassification, as indicated by its deployments by Secunet [29,39] and as highlighted by our case study.

2 Attacks

We begin by providing evidence that SPARK's flow analysis is termination-, progress-, and timing-insensitive. We do this by providing minimal example programs which pass analysis yet leak information. Since SPARK's flow analysis implementation has no proof of soundness, this helps us identify the property it is meant to enforce, and thus how to improve it.

All our examples share the same structure: a *Main* file that reads a byte from standard input, and invokes a procedure `Leak` on said byte.

```
procedure Leak (H : in out Byte)
  with Global  => (In_Out => Standard_Output),
       Depends => (H => H, Standard_Output => Standard_Output);
```

This specification states that `Leak` performs I/O on its parameter `H` and the global `Standard_Output`, and that output on `Standard_Output` *only* depends on `Standard_Output`. That last bit is the *flow policy* of `Leak`. Our attacks, which differ only in how they implement `Leak`, aim to violate this flow policy while passing analysis, by making output on `Standard_Output` depend on `H`.

The source code for our attacks is in Appendix A. In this section, we focus on the two attacks most relevant to our technical contributions (termination and progress), and summarize the other attacks when closing the section.

Termination. A flow analysis is *termination sensitive* when it tracks whether a value can affect termination behavior. To gauge whether SPARK's flow analysis is termination sensitive, we design `Leak` on the left of Figure 1 such that the *presence of output* on standard output depends on whether the program enters an infinite loop, which depends on `H`. *Main* passes analysis with this implemetation of `Leak`. However, invoking `Leak` on input values 1 and 2 produces different

observable behavior: with input 1, we see '!' on the standard output; with input 2, *Main* diverges. Thus, SPARK's flow analysis is *termination insenstive*.

Progress. A termination insensitive flow analysis permits one bit to leak through termination observations. Programs that pass such an analysis can leak much more when a value can affect the progress the program makes on producing its intermediate output [7]. A flow analysis that tracks such flows is *progress sensitive*. The right panel of Figure 1 shows the brute force attack by Askarov et al. [7] modelled in SPARK. Here, `Leak` outputs on standard output all characters in their ASCII number order up to the character numbered `H`, and diverges, thus leaking all of `H`. Again, this program passes SPARK's flow analysis, indicating that the flow analysis is *progress insensitive*.

Summary. We studied SPARK's flow analysis under three additional attacks:

- *Resource Exhaustion:* We replace nontermination in the progress attack with abnormal termination (a stack overflow). We give two examples; one allocates an array too large to fit on the stack, the other creates infinitely many stack frames through infinite mutual recursion. Both pass analysis.
- *Timing:* A flow analysis is *timing sensitive* when it tracks whether a value can affect the time an effect occurs. We replace nontermination in the termination attack with a computation which takes considerable time (selection-sorting 2^{16} bytes). The attack passes analysis.
- *Explicit and Implicit flows:* As a sanity check, we provide two implementations of `Leak`: one creates an explicit flow from `H` to `Standard_Output`, the other an implicit flow. The attacks do *not* pass analysis.

Since SPARK detected the explicit and implicit flows, but failed to detect our other attacks, it appears that SPARK enforces progress-insensitive security. As demonstrated above, whole secrets can leak through progress. In this paper, we harden SPARK's flow analysis to detect progress leaks, to enforce progress sensitive security. Addressing the other attacks is out of scope of this paper.

3 Programs and policies

We explain our ideas and results using a simple while language with flow annotations, inputs, outputs, and arrays, which is a stripped down version of SPARK. For a formal semantics and illustrative examples, see Appendix B and C.

Programs. The syntax for our language is given in Figure 2. Let p range over programs, b over blocks, x over array names, e over expressions, n over integers, c over channels, and $\odot$ over (total) binary integer operators. Here, $x\texttt{[}e\texttt{]}$ denotes index e in array x. To model non-array variables, we write x as syntactic sugar for $x\texttt{[0]}$. Statement c `<-` e outputs integer e to channel c, and c `->` $x\texttt{[}e\texttt{]}$ inputs an integer on c and stores it in $x\texttt{[}e\texttt{]}$. The rest is standard.

Control flow graphs. A control flow graph (CFG) represents a program as a directed graph. The CFG of a program p is defined by $\rightarrow$ in Figure 2; p' is a node iff $p \rightarrow^* p'$, and (p', p'') is an edge iff p' and p'' are nodes and $p' \rightarrow p''$.

$$
\begin{array}{lll}
p ::= \texttt{skip} & b ::= \texttt{skip} & \\
\quad \mid b\texttt{;}\ p & \quad \mid x\texttt{[}e\texttt{]}\ \texttt{:=}\ e & \\
 & \quad \mid c\ \texttt{<-}\ e & \\
e ::= n & \quad \mid c\ \texttt{->}\ x\texttt{[}e\texttt{]} & \\
\quad \mid x\texttt{[}e\texttt{]} & \quad \mid \texttt{if}\ e\ \texttt{\{}p\texttt{\}}\ \texttt{\{}p\texttt{\}} & \\
\quad \mid e \odot e & \quad \mid \texttt{while}\ e\ \texttt{\{}p\texttt{\}} &
\end{array}
$$

$$
\begin{array}{l}
\texttt{if}\ e\ \texttt{\{}p_1\texttt{\}}\ \texttt{\{}p_0\texttt{\};}\ p \to p_1\texttt{;}\ p \\
\texttt{if}\ e\ \texttt{\{}p_1\texttt{\}}\ \texttt{\{}p_0\texttt{\};}\ p \to p_0\texttt{;}\ p \\
\texttt{while}\ e\ \texttt{\{}p_1\texttt{\};}\ p \to p_1\texttt{;}\ \texttt{while}\ e\ \texttt{\{}p_1\texttt{\};}\ p \\
\texttt{while}\ e\ \texttt{\{}p_1\texttt{\};}\ p \to p \\
\text{For } b \in \{\texttt{skip}, x\texttt{[}e\texttt{]}\ \texttt{:=}\ e', c\ \texttt{<-}\ e,\ c\ \texttt{->}\ x\texttt{[}e\texttt{]}\}\text{:} \\
\quad b\texttt{;}\ p \to p
\end{array}
$$

Fig. 2: Program syntax (left) and CFG (right)

We distinguish two nodes in the CFG of program p: the START node p and the END node `skip`. START is defined as the root of the graph. END has no outgoing edges. Conventionally, CFG nodes are blocks, b. This representation is obtained by dropping p from nodes of the form $b\texttt{;}\ p$ and replacing $\texttt{if}\ e\ \texttt{\{\_\}}\ \texttt{\{\_\}}$ and $\texttt{while}\ e\ \texttt{\{\_\}}$ nodes with $\texttt{branch}\ e$. See Appendix C for an illustrative CFG.

Semantics. A program executes in a memory $m : \mathbb{X} \times \mathbb{N} \to \mathbb{Z}$, which provides a (mutable) binding for every location of every array (initially all set to 0 in the initial memory m_0), and an environment $\boldsymbol{e} : \mathbb{C} \to \mathbb{Z}^\omega$, which provides an infinite stream of input values on every channel. We use a small-step reduction relation $(\boldsymbol{e}, m, p) \xrightarrow{o} (\boldsymbol{e}', m', p')$. Here, $o ::= \bullet \mid !cv$ is the *output* of the reduction step; if $p = c\ \texttt{<-}\ e\texttt{;}\ p'$, then $o = !cv$ where v is the value e evaluates to; otherwise, $o = \bullet$. The full definition of $\xrightarrow{o}$ is shown in Appendix B. Let $\bar{o} = o_1 \ldots o_n$ denote a finite sequence of outputs, and let $\xrightarrow{\bar{o}} = (\xrightarrow{\bullet})^*\ \xrightarrow{o_1}\ (\xrightarrow{\bullet})^* \ldots (\xrightarrow{\bullet})^*\ \xrightarrow{o_n}\ (\xrightarrow{\bullet})^*$.

Our environments are *total* [32], i.e., never block output, and always provide input on request. This is a natural fit for SPARK, as safety-critical systems typically perform nonblocking I/O (e.g. on files and POSIX shared memory using `read()` and `write()` from the Single UNIX Specification). The endpoints of channels thus, in general, form a collective store which can change independently of the program, and provide input that depends on past output. However, Clark and Hunt [15] have shown that *when reasoning about security* of deterministic programs (as in our case), environments can be simplified to streams. We use this simplification here. Programs can be composed securely under these environments as long as their scheduler is secure and deterministic. For a more complete and general treatment of composition, see [32,33].

Flow policies. A flow policy expresses permitted flows between input and output channels. We are interested in two kinds of dependencies: where input affects the *presence* (i.e. occurrence) resp. *content* (i.e. value) of an output. The syntax of our flow policy language is given in Figure 3. Let f range over flow policy specifications, and d over dependencies. The syntax $c\ \texttt{=>}\ c'$ (resp. $c\ \texttt{->}\ c'$) means that content (resp. presence) of output on c is allowed to depend on input on c'. For instance, a flow policy stating that (only) the presence of output on StdErr (standard error) is allowed to depend on input on StdIn (standard input) can be written as `StdErr -> StdIn; null`. Every flow policy f straightforwardly yields a pair of functions (π, κ) where $\pi(c)$ (resp. $\kappa(c)$) is the set of input channels on which the presence (resp. content) of output on c may depend. We lift these functions to sets of channels: $\pi(C) = \bigcup_{c \in C} \pi(c)$ and $\kappa(C) = \bigcup_{c \in C} \kappa(c)$.

$$
\begin{array}{ll}
f ::= \texttt{null} & d ::= c\ \texttt{=>}\ c \\
\quad \mid d\texttt{;}\ f & \quad \mid c\ \texttt{->}\ c
\end{array}
$$

Fig. 3: Syntax of flow policies

4 Security property

Consider a fixed policy (π, κ). Our attackers observe all outputs on some output channels. An attacker or *observer* $\omega = (\omega_\pi, \omega_\kappa)$ is a pair where ω_π (resp. ω_κ) is the set of channels on which the presence (resp. content) of outputs is observed. If an observer sees the content of outputs on a channel, it can certainly detect the presence of outputs on the channel, so we require $\omega_\kappa \subseteq \omega_\pi$. Two environments are equivalent to an observer ω if the environments agree on all input channels that may flow to outputs visible to ω.

Definition 1 (ω-equivalence of e). *$\boldsymbol{e}$ and $\boldsymbol{e}'$ are ω-equivalent, $\boldsymbol{e} \sim_\omega \boldsymbol{e}'$, iff* $\forall c \in \pi(\omega_\pi) \cup \kappa(\omega_\kappa) \,.\, \boldsymbol{e}(c) = \boldsymbol{e}'(c)$.

The observables in an output are defined as follows: $!cv\restriction_\omega = !cv$ if $c \in \omega_\kappa$, $!c\mathtt{d}$ if $c \in \omega_\pi \setminus \omega_\kappa$, and $\bullet$ otherwise (here `d` is a default output, like `null` or `0`). We remove the unobservables of a sequence of outputs $\bar{o}$ follows: $\epsilon\restriction_\omega = \epsilon$, $(o.\bar{o})\restriction_\omega = \bar{o}\restriction_\omega$ if $o\restriction_\omega = \bullet$, and $(o\restriction_\omega).(\bar{o}\restriction_\omega)$ otherwise.

Definition 2 (ω-equivalence of $\bar{o}$). *$\bar{o}$ and $\bar{o}'$ are ω-equivalent, $\bar{o} \sim_\omega \bar{o}'$, iff* $\bar{o}\restriction_\omega = \bar{o}'\restriction_\omega$.

Our security property, *progress-sensitive noninterference* (PSNI), requires that under observably equivalent environments, a program must be able to componentwise observably-equivalently match observable outputs in its behaviors [31,8,12,32]. For an example involving PSNI, see Appendix C.

Definition 3 (Progress-sensitive noninterference). *p satisfies* PSNI *iff* $\forall \omega, \boldsymbol{e}, \boldsymbol{e}' \,.\, \boldsymbol{e} \sim_\omega \boldsymbol{e}' \implies \forall \bar{o} \,.\, (\boldsymbol{e}, m_0, p) \xrightarrow{\bar{o}} \implies \exists \bar{o}' \,.\, (\boldsymbol{e}', m_0, p) \xrightarrow{\bar{o}'} \wedge\, \bar{o} \sim_\omega \bar{o}'$.

5 Enforcement

SPARK implements a dependency analysis on control flow graphs that prevents all explicit and implicit information leaks, but does not prevent leaks due to progress and termination. In this section, we explain how to augment such a dependency analysis with a loop termination oracle to enforce the stronger property progress-sensitive noninteference (PSNI, Definition 3). While loop termination oracles have been combined with type sytems to enforce PSNI in prior work (e.g., [28]), our technical development makes three novel contributions: (1) We use a graph-based analysis to enforce PSNI (2) Our dependency analysis handles reactive programs, and (3) Our dependency analysis accounts for the difference between output content and output presence. In the following, we describe our analysis for the core language from Section 3 and prove that it enforces PSNI. The core language captures all essential features of SPARK, so generalizing the analysis to all of SPARK should not be difficult.

Standard data- and control-dependency analysis. SPARK's flow analysis uses standard dependency analysis [20,26], which we review briefly. We say that

a node b in a CFG reads array x if b contains x in at least one location other than x`[...] := ...`. Dually, b writes to array x if $b = (x$`[`e`] :=` $e')$. Node b reads a channel c if $b = (c$ `->` `...[...]`$)$. Dependency analysis outputs all the nodes of the CFG on which a given node is *data dependent* or *control dependent*. Data dependence arises due to data flow. E.g., in `x = 1; y = 3; z = x + 2; a = z`, the statements `z = x + 2` and `a = z` are data dependent on the statement `x = 1`, but not on `y = 3`. Similarly, in the example of Figure 1 (right), the statements on lines 5 and 6 (output and branch `K >= H`, respectively) are data dependent on the statement `K := K + 1` on line 11.

Definition 4 (Data dependence). *A node b is data dependent on node b' in a CFG G, written $dd_G(b', b)$, if there is a path $b' \rightarrow^* b \in G$ and there is an array that b' writes and b reads, or there is a channel that both b and b' read.*

Note that the statement in b does not have to be an assignment; the definition implies a data dependence from `x = y` to `c <- x` in program `x = y; c <- x`. Also, as commonly assumed by flow analyses in prior work, e.g. Jif [30] and Paragon [14], our definition of data dependence is *flow-insensitive*. This means it ignores the effects of writes in nodes strictly between b' and b; in program `x = y; x = 0; z = x`, node `z = x` is data dependent on the node `x = y` by our definition, even though `x` is overwritten by a constant between the nodes. (We use some lemmas from [1] in our proofs, but this difference does not impact those lemmas.) For clues on how to make this definition flow-sensitive, see [23].

Control dependence captures influence due to branches. In the program `if (x > 0) { y = 1 } else { y = 2 }; z = 1`, both the nodes `y = 1` and `y = 2` are control dependent on the branch node `x > 0`. However, the node `z = 1` is *not* control dependent on `x > 0` because it executes irrespective of the outcome of the test `x > 0`. There are many different definitions of control dependence in literature (see [34] for a survey). We define here the most standard notion of control dependence, which suffices for our purposes. We say that node b *post-dominates* b' if every path from b' to END passes through b.

Definition 5 (Control dependence [1]). *A node b is control dependent on node b' in a CFG G, written $cd_G(b', b)$, if the following hold: (1) Either $b = b'$ or b does not post-dominate b' in G, and (2) There is a nontrivial path $b_1 \rightarrow \ldots \rightarrow b_k \in G$ with $b_1 = b'$, $b_k = b$ such that for all $i \in 2 \ldots k-1$, b post-dominates b_i.*

For block-structured languages such as SPARK and the core calculus of Section 3, a node b is control dependent on node b' iff b is a branch or loop condition and b' lies within that branch or loop. However, control dependence is defined on arbitrary CFGs, even those without block structure (we exploit this generality later). Combining data- and control-dependency analysis, we define *dependence* as the reflexive-transitive closure of the data- and control-dependence relations. For example, in the program of Figure 1 (right), the `while` loop on line 7 is dependent on the statement `K := K + 1` on line 11 because the condition `K >= H` on line 6 is data-dependent on line 11, and line 7 is control-dependent on line 6. The set of all nodes on which a node b depends is called b's backward slice.

Definition 6 (Dependence and backward slice). *The dependence relation dep_G for CFG G is defined as $(dd_G \cup cd_G)^*$. The backward slice of node b, $BS_G(b) = \{b' \mid dep_G(b', b)\}$, is the set of all nodes on which b is dependent.*

Information flow control using dependency analysis. The dependence relation dep_G captures all explicit and implicit flows, and, hence, can be used for enforcement of information flow policies. There are well-known algorithms to compute dependencies and backward slices efficiently, e.g., [26]. This analysis is already implemented in SPARK. However, noninterference enforced this way is progress-*in*sensitive because the dependency analysis described above does not take into account nonterminating loops. For instance, the program of Figure 1 (right) passes SPARK's dependency analysis, even though it leaks `H` to a progress-sensitive adversary who can observe `K`. Additionally, the method so far has been limited to sequential programs where the adversary makes only one observation at the end of the program. We explain how the method can be adapted to enforce progress-sensitive noninterference on reactive programs, additionally accounting separately for output content and output presence.

Progress-sensitive dependence. A leak due to progress happens when an attacker-visible output is pre-empted due to the nontermination of a branch with a secret branch condition. Our simple insight is that such leaks can be detected by a dependence analysis if we ensure the following:

Requirement 1. An output that can be reached *after* the end of a branch is dependent on the branch point if some loop in the branch may diverge.

To implement Requirement 1, we use a static termination analysis, often called a termination oracle [17,27,45]. This oracle determines which loops in the program may diverge. We add an edge from every node in such a loop to the END node of the CFG. It is easy to check that the modified CFG satisfies Requirement 1 if the termination oracle is *sound*, i.e., it flags all loops that diverge on some input. A trivial, sound termination oracle marks every loop as potentially non-terminating. The use of this oracle in our analysis causes every program that contains an attacker-visible output after a loop with a secret loop condition to be marked as leaky, irrespective of whether or not the loop diverges, which may result in false positives. This corresponds exactly to termination-sensitive analyses developed by Smith [41] and Boudol and Castellani [13]. False positives can be reduced using a more precise termination oracle. For example, the program `while (h <> h) { }; l = 1` does not have a flow (via progress or otherwise) from the input variable `h` to the output variable `l`, but the trivial oracle above will cause this program to be marked leaky by the analysis. On the other hand, a slightly better oracle that uses symbolic analysis to infer that `(h <> h)` is always `false` will cause the program to be accepted. In general, fewer false positives in the termination oracle translate to fewer false positives in our dependence analysis. Consequently, we present our analysis *parametrically* in the termination oracle, leaving it to the specific implementation to decide how many resources to devote to the oracle (and, hence, how much precision to obtain).

Definition 7 (Termination oracle). *A termination oracle T is a function that maps a CFG to a subset of the CFG's nodes. T is sound if for every CFG G and every node $b \in G$, $b \in T(G)$ if there is a memory m and environment $\boldsymbol{e}$ such that b appears infinitely often in the reduction sequence starting from the state $(\boldsymbol{e}, m, \mathsf{START})$.*

Definition 8 (Progress-sensitive graph). *Given a control flow graph G and a termination oracle T, the progress-sensitive CFG $ps_T(G)$ is defined by adding to G the edges $\{(b, \mathsf{END}) \mid b \in T(G)\}$.*

For the program of Figure 1 (right), a sound termination oracle T will say that the `while` loop on line 7 is nonterminating and, hence, $\mathrm{ps}_T(G)$ will contain an edge from the branch condition of the loop to the end of the program. This makes the output statement on line 5 dependent on the branch condition `K >= H` and, hence, a dependency analysis will discover the progress leak in the program. Note that $\mathrm{ps}_T(G)$ may not correspond to any block-structured program.

Enforcing PSNI with content and presence distinction. Our analysis takes as input a policy f and a program p. It works as follows. Let G be p's CFG. We compute the progress-sensitive CFG $G' = \mathrm{ps}_T(G)$. Then, for each node $b \in G'$ that outputs to some channel c, we compute the backward slice of b in G', and check that the policy relation κ allows a flow to c from any channel c' on which an input is made in the backward slice. This ensures that information flows to the *content* of messages on c only in accordance with the policy. To account for flows due to *presence* of outputs on c, we compute a second backward slice from the same node b, but after erasing the payload of the output in b. We check that the policy relation π (not κ) allows a flow to c from any channel c' on which an input is made in this backward slice. Thus, by computing two backward slices per output node, we capture separate observations of content and presence. In the sequel, we assume a fixed policy $f = (\kappa, \pi)$.

Definition 9 (Enforcement of PSNI). *Let p be a program with CFG G. Let $G' = ps_T(G)$. We say p passes the* PSNI *enforcement, written $\mathsf{check}_T(p)$, if the following hold for any node b of the form c `<-` e in G':*

1. *If c' `->` x `[`e'`]` $\in BS_{G'}(b)$ then $c' \in \kappa(c)$.*
2. *If c' `->` x `[`e'`]` $\in BS_{\hat{G}'}(\hat{b})$ then $c' \in \pi(c)$, where $\hat{G}'$ is obtained by replacing b with $\hat{b} = c$ `<-` `d` in G'.*

Our main theorem is that the enforcement above is sound: If $\mathsf{check}_T(p)$, then p satisfies PSNI. We prove the theorem using bisimulations on backward slices [1]. Our proof is inspired by a related proof for enforcement of progress-insensitive noninterference in a sequential language [51]. In contrast to that proof, our proof captures progress-sensitive noninterference for a reactive language. Handling reactivity is quite involved: With multiple outputs, we have to argue that the *order* of observable outputs (at different program points) is independent of secret inputs. To do this, we construct a hypothetical slice that is the union of slices from all outputs visible to a given adversary. See Appendix D for details.

```
0  procedure Leak (H : in out Byte) is
     E : exception; -- new exception
     X : Byte := 0;
     K : Byte := 0;
     O : File_Type := Standard_Output;
5  begin
     H := H;
     if X = 1 then raise E; end if;
     while True loop

10     Write (O, K);
       if K >= H then
         if X = 1 then raise E; end if;
         while True loop

15         H := H;
         end loop;
       end if;
       K := K + 1;
     end loop;
20 end Leak;
```

```
0  procedure Leak (H : in out Byte) is
     E : Byte := 0; -- E := 1 when an
                    -- exception is raised
     K : Byte := 0;
     O : File_Type := Standard_Output;
5  begin
     if E = 0 then H := H; end if;
     if E > 0 then E := 1; end if;
     while E = 0 and then True loop
       E := E;
10     if E = 0 then Write (O, K); end if;
       if E = 0 and then K >= H then
         if E > 0 then E := 1; end if;
         while E = 0 and then True loop
           E := E;
15         if E = 0 then H := H; end if;
         end loop;
       end if;
       if E = 0 then K := K + 1; end if;
     end loop;
20 end Leak;
```

Fig. 4: Source-to-source transformation of the program in the left of Figure 1, using exceptions (left) and using an emulation of exceptions (right)

Theorem 1 (Soundness of enforcement). *If T is a sound termination oracle and* $\mathsf{check}_T(p)$*, then p satisfies* PSNI.

6 Source-to-source transform

The previous section describes a CFG transformation which ensures Requirement 1 – that any outputs after a potentially divergent branch depend on the branch's condition, which is used to enforce PSNI. In this section, we describe a source-to-source transform that also implies Requirement 1. The transform can be used to enforce PSNI using a standard, *unmodified* (and, hence, closed-source) dependency analysis of the kind that exists for SPARK.

The goal of our source-to-source transform, like the CFG transform, is to add a direct path from every potentially infinite loop to the end of the program. If the existing dependency analysis supports programmatic exceptions, then the transform is trivial: Just before every potentially infinite loop (identified by the oracle T), we add a statement to raise an unhandled exception, conditional on an unsatisfiable predicate. This has the effect of simulating an edge from the loop to the end of the program because the exception is not handled anywhere. It is quite easy to see that this has the same effect as the CFG transform. For example, the program in the right of Figure 1 would be transformed to the program in the left of Figure 4. Observe the new line 12 with the `raise` statement.

If the flow analysis, like SPARK, does not track flows through exceptions[1], then the source-to-source transform can emulate an exception by adding a new

[1] The SPARK 2014 documentation states that SPARK programs are allowed to raise exceptions, but may not handle them. However, in our experiments with SPARK GPL 2015, we found that the flow analysis did not track flows through exceptions.

boolean variable, say `E`, initially set to `0`. `E` is set to `1` where the exception is to be raised, and the program is transformed to check that `E` is still `0` before executing any statement or entering any branch of the original program. This ensures that once the exception is "raised" (`E` is set to `1`), no statement from the original program executes and control propagates to the end of the program silently. This transform can be defined formally, but we only illustrate it for the progress leak in Figure 1 in the right panel of Figure 4. Observe that there is now a dependency between the branch condition `K >= H` and the output statement on line 10.

We note that to enforce PSNI, the dependency analysis should be applied directly to the output of either of the two transforms described above, without any intervening compiler optimizations. Such optimizations can negate the effects of our transforms. For instance, constant propagation followed by dead code elimination would remove the two `raise` statements on lines 7 and 12 in the left panel of Figure 4 and, hence, also remove the control dependencies introduced deliberately by the transform.

7 Case Study

We demonstrate the usefulness of our approach on a nontrivial application by implementing a control system for a cruise missile (derived from publicly available code by Hilton [25]), and applying our approach on the code to prove desired properties. The code steers the missile towards a target coordinate, and detonates a nuclear warhead once within range, or self-destructs in the event that a device fails. The code is intended to be an illustrative model native to the domain of SPARK. The code makes several simplifications (e.g. the missile flies in 2D space), and there are many safety- and mission-critical considerations for more realistic missile control systems that we have not considered. For details on such considerations, see Hilton [25]. We give an overview of our case study (all our code is online [49]).

The missile has three sensors: a *failure detector*, which reports when a device has failed; an *intertial navigation system*, which provides spatial orientation and displacement readings (via accelerometers and a ring laser gyroscope) for navigation by dead reckoning; and a *clock*, used to calculate orientation and displacement from accelerometer readings through integration. Using these readings, the code controls three actuators: a *watchdog*, which, if not actuated at regular intervals, triggers self-destruction (to avoid unwanted consequences of device or software failure); a *nuclear warhead*, which is detonated when the missile reaches its target; and *steering*, consisting of aerodynamic fins which the code actuates for trajectory corrections. Architecturally, our code draws inspiration from an existing case study on implementing a controller for a water boiler [43, Section 7]. The *Main* module consists of a sense-control-actuate loop, in which it commands the sensor modules to read from their device, uses these readings to compute values to control the actuators, and invokes the actuator modules to actuate their device. The inter-module information flows are given in Figure 5.

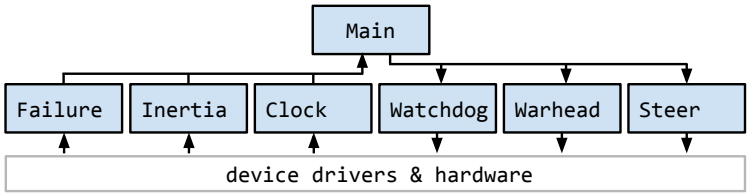


Fig. 5: Inter-module information flows in the missile control system

To illustrate our approach, we aim to prove that orientation does not affect self-destruction. The body of the *Main* procedure, in the left of Figure 6, is of primary concern. Without our approach, since SPARK assumes loops terminate, reaching "`Watchdog.Actuate`" in each iteration is deemed inevitable by SPARK, so SPARK (incorrectly) claims the presence of a destruct event does not depend on *any* input. However, if we instead apply the SPARK analysis on the code resulting from applying our source-to-source analysis from Section 6 on the missile control system source code, we get a different result: SPARK (correctly) infers that the presence of a destruct event depends *only* on device failure. This can be seen by inspecting the result of the transformation of the main loop, in the right of Figure 6. Since both loops are of the form "`while True loop`", any sound termination oracle would flag them both as possibly diverging. Hence we emulate a raised exception before both loops, and add a check on variable `E` to each branch. SPARK no longer deems that reaching "`Watchdog.Actuate`" is inevitable; it now depends on the value of `E`. SPARK deems that the value of `E` depends on `Destruct`, since there is an assignment to `E` under a branch on `Destruct`. Since `Destruct` depends *only* on device failure, and since the only other assignment to `E` branches only on `E`, self-destruction depends only on device failure.

8 Related Work

We focus on the three most closely related areas of work: information flow tools, progress-sensitive security, and information flow analysis for SPARK.

Information flow tools. As mentioned before, much progress has been made on enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as FlowFox [22], JSFlow [24] and IFC4BC [10] for JavaScript, Jif [30], Paragon [14] and JOANA [23] for Java, FlowCaml [40] for Caml, LIO [47] for Haskell, and SPARK flow analysis [9] for SPARK. With the exception of the latest versions of LIO, these tools target *progress-insensitive noninterference* [7,8,12], allowing secrets to affect progress of public computation. With the focus on the termination and timing channels, Stefan et al. [46] introduced restrictions in LIO on side effects that follow secret branching, which help enforce stronger policies.

Progress-sensitive security. *Progress-sensitive noninterference* [31,8,12,32,6] (PSNI) disallows progress leaks. PSNI is not susceptible to laundering secrets by brute-force attacks [7] or re-running programs [11]. A typical approach to enforcing PSNI is to disallow loops with secret guards, going back to Volpano

```
while True loop
   -- [...] (sense, control)
   Steer.Actuate;
   if Destruct then
      -- block watchdog.
      while True loop
         null;
      end loop;
   end if;
   Watchdog.Actuate;
   if Detonate then
      Warhead.Actuate;
   end if; -- [...]
end loop;
```

```
if E > 0 then E := 1; end if;
while E = 0 loop
   E := E; -- [...] (sense, control)
   if E = 0 then Steer.Actuate; end if;
   if E = 0 and then Destruct then
      if E > 0 then E := 1; end if;
      while E = 0 loop
         E := E; null;
      end loop;
   end if;
   if E = 0 then Watchdog.Actuate; end if;
   if E = 0 and then Detonate then
      Warhead.Actuate;
   end if; -- [...]
end loop;
```

Fig. 6: Main loop, before (left) and after (right) transformation

and Smith's technique to deal with termination leaks [48], or to allow loops with secret guards but prohibit assignments to public variables that follow such loops [41,13]. While the theory of progress-sensitive security has been explored [31,8,12,32,6], our work connects the theory with tools, showing how we can leverage a progress-insensitive tool (SPARK's flow analysis) to achieve PSNI. Related to our source-to-source transform, Russo et al. [36] discuss *magnification patterns* in the context of distinguishing flows in malicious and nonmalicious code. A magnification pattern in a control-flow graph consists of a branching on a secret guard inside of a loop. We note that in the absence of such patterns (as is sometimes the case in nonmalicious code [36]), progress-sensitive security and progress-insensitive security coincide. Moore et al. [28] use *termination oracles* for termination-sensitive tracking. Their prototype implementation utilizes an SMT solver to analyze examples in a simple imperative language. While related, there are several distinguishing features of our work: we focus on practical information flow control in SPARK and push the approach to the full SPARK language; our case study goes beyond code snippets to a suite for a missile controller; on the theoretical side, our framework is graph-based, which opens up possibilities for natural adoption to other graph-based tools such as JOANA [23].

Information flow analysis in SPARK. A line of work by Amtoft et al. shares with our work the motivation to improve SPARK's information flow analysis. Based on an expressive information logic [2], they enhance the information flow contract language to support compositional policies and conditional information flows [5]. They improve the precision of the analysis by breaking out of a limitation of the original analysis that treats arrays as indivisible entities and evaluate the approach on a collection of SPARK programs [4]. They extend the logical framework to produce machine-checkable formal certificates of correctness for verified code [3]. Extending the results by Amtoft et al. to guarantee progress-sensitive security is a promising direction for future work.

9 Conclusion

This paper puts a spotlight on the SPARK flow analysis. Articulating the boundaries of what is achievable by the analysis, we spell out the attacks to exploit

such channels as termination, progress, resource exhaustion, and timing channels. We suggest how to harden the analysis to achieve security against stronger attackers, with the focus on progress-sensitive security as our baseline. Instead of redesigning and reimplementing the enforcement, we show how to leverage known flow analyses for weaker attackers by a transform on program dependence graphs. The graph transform represents termination and progress flows by injecting additional edges. We establish the soundness of this approach for a core language and demonstrate that it can be applied as a source-to-source transform of SPARK code when modifying the compiler is undesirable. A case study with a control unit of a missile written in SPARK 2014 indicates the usefulness of the approach. Future work is focused on enriching the policy and enforcement mechanisms with possibilities for *declassification* [38], a feature on the wish list of the SPARK developers [35]. We are also interested in extending the framework with treating resource exhaustion and timing leaks and exploring more sophisticated attacks.

Acknowledgments Thanks are due to Angela Wallenburg for inspiration and regular updates about developments on SPARK. This work was funded by the European Community under the ProSecuToR and WebSand projects, the Swedish research agencies SSF and VR and the German DFG priority program "Reliably Secure Software Systems" (RS3).

References

1. Amtoft, T. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processing Letters 106*, 2 (2008), 45 – 51.
2. Amtoft, T., Bandhakavi, S., and Banerjee, A. A logic for information flow in object-oriented programs. In *POPL* (2006), pp. 91–102.
3. Amtoft, T., Dodds, J., Zhang, Z., Appel, A. W., Beringer, L., Hatcliff, J., Ou, X., and Cousino, A. A certificate infrastructure for machine-checked proofs of conditional information flow. In *POST* (2012), pp. 369–389.
4. Amtoft, T., Hatcliff, J., and Rodríguez, E. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In *ESOP* (2010), pp. 43–63.
5. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., and Greve, D. Specification and checking of software contracts for conditional information flow. In *FM* (2008), pp. 229–245.
6. Askarov, A., Chong, S., and Mantel, H. Hybrid Monitors for Concurrent Noninterference. In *CSF* (July 2015).
7. Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proc. European Symp. on Research in Computer Security* (Oct. 2008), vol. 5283 of *LNCS*, Springer-Verlag, pp. 333–348.
8. Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium* (July 2009).
9. Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

10. Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Information flow control in webkit's javascript bytecode. In *Proc. Principles of Security and Trust (POST)* (2007), pp. 159–178.
11. Birgisson, A., and Sabelfeld, A. Multi-run Security. In *ESORICS* (2011), pp. 372–391.
12. Bohannon, A., Pierce, B., Sjöberg, V., Weirich, S., and Zdancewic, S. Reactive Noninterference. In *ACM Conference on Computer and Communications Security* (Nov. 2009), pp. 79–90.
13. Boudol, G., and Castellani, I. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science 281*, 1 (June 2002), 109–130.
14. Broberg, N., van Delft, B., and Sands, D. Paragon for Practical Programming with Information-Flow Control. In *APLAS* (2013), C. chieh Shan, Ed., vol. 8301 of *Lecture Notes in Computer Science*, Springer, pp. 217–232.
15. Clark, D., and Hunt, S. Noninterference for Deterministic Interactive Programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)* (Oct. 2008).
16. Cohen, E. S. Information transmission in sequential programs. In *Foundations of Secure Computation*, R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, Eds. Academic Press, 1978, pp. 297–335.
17. Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI* (2006), pp. 415–426.
18. Denning, D. E. A lattice model of secure information flow. *Comm. of the ACM 19*, 5 (May 1976), 236–243.
19. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Comm. of the ACM 20*, 7 (July 1977), 504–513.
20. Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (1987), 319–349.
21. Goguen, J. A., and Meseguer, J. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy* (Apr. 1982), pp. 11–20.
22. Groef, W. D., Devriese, D., Nikiforakis, N., and Piessens, F. Flowfox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security* (2012).
23. Hammer, C., and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security 8*, 6 (Dec. 2009), 399–422.
24. Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. 29th ACM Symposium on Applied Computing* (2014).
25. Hilton, A. J. *High Integrity Hardware-Software Codesign*. PhD thesis, The Open University, April 2004.
26. Horwitz, S., Reps, T. W., and Binkley, D. Interprocedural slicing using dependence graphs. In *PLDI* (1988), pp. 35–46.
27. Kroening, D., Sharygina, N., Tsitovich, A., and Wintersteiger, C. M. Termination analysis with compositional transition invariants. In *CAV* (2010), pp. 89–103.
28. Moore, S., Askarov, A., and Chong, S. Precise enforcement of progress-sensitive security. In *ACM Conference on Computer and Communications Security* (2012), pp. 881–893.
29. The Muen Separation Kernel. `http://muen.codelabs.ch/`.

30. Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.
31. O'Neill, K., Clarkson, M., and Chong, S. Information-flow Security for Interactive Programs. In *Proc. IEEE Computer Security Foundations Workshop* (July 2006), pp. 190–201.
32. Rafnsson, W., Hedin, D., and Sabelfeld, A. Securing Interactive Programs. In *Proc. IEEE Computer Security Foundations Symposium* (June 2012).
33. Rafnsson, W., and Sabelfeld, A. Compositional Security for Interactive Systems. In *CSF* (2014), pp. 277–292.
34. Ranganath, V. P., Amtoft, T., Banerjee, A., Hatcliff, J., and Dwyer, M. B. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems 29*, 5 (Aug. 2007).
35. Refined Information Flow Requirement. http://lists.forge.open-do.org/pipermail/spark2014-discuss/2012-December/000682.html.
36. Russo, A., Sabelfeld, A., and Li, K. Implicit flows in malicious and nonmalicious code. *2009 Marktoberdorf Summer School (IOS Press)* (2009).
37. Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications 21*, 1 (Jan. 2003), 5–19.
38. Sabelfeld, A., and Sands, D. Declassification: Dimensions and principles. *J. Computer Security 17*, 5 (Jan. 2009), 517–548.
39. Multilevel Workstation: High-Security Framework, Pilot, and Formalization Architecture. http://www.secunet.com/fileadmin//sina_downloads/Produktinfo_englisch/SINA-Multilevel_Brochure_en.pdf.
40. Simonet, V. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.
41. Smith, G. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop* (June 2001), pp. 115–125.
42. SPARK (programming language). http://en.wikipedia.org/wiki/SPARK_%28programming_language%29.
43. Tool Development and Support: INFORMED Design Method for SPARK. http://docs.adacore.com/sparkdocs-docs/Informed.htm.
44. SPARK 2014. http://www.spark-2014.org/.
45. Spoto, F., Mesnard, F., and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.* (2010).
46. Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., and Mazières, D. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP* (2012), pp. 201–214.
47. Stefan, D., Russo, A., Mitchell, J., and Mazières, D. Flexible dynamic information flow control in haskell. In *Proc. Haskell Symposium* (2011), ACM, pp. 95–106.
48. Volpano, D., and Smith, G. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop* (June 1997), 156–168.
49. W. Rafnsson and D. Garg and A. Sabelfeld. Progress-Sensitive Security for SPARK. Full version: http://research.precise.li/pub/2016essos
50. Walker, D. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, 2005, ch. 1.
51. Wasserrab, D., Lohner, D., and Snelting, G. On PDG-based noninterference and its modular proof. In *PLAS* (2009), pp. 31–44.

A Attacks

Here we outline the structure, and provide the full source code, for the attacks presented Section 2. This code can also be downloaded [49]. The structure consists of three modules: a *Main* module containing the procedure which gets compiled to an executable, a *Leak* module which provides a procedure for the main module to leak information, and an *Aux* module which provides a simplified interface to system I/O.

The *Main* module, `main.adb`, consists of a single procedure `Main` which reads from standard input by invoking the auxiliary module *Aux*, and leaks said input to standard output by invoking the leak module *Leak*. Its source follows.

```
pragma SPARK_Mode (On);                                    -- File: main.adb
with SPARK.Text_IO,Aux,Leak; use SPARK.Text_IO,Aux,Leak;

procedure Main
  with Global  =>
         (In_Out => (Standard_Input, Standard_Output, Standard_Error)),
       Depends =>
         (Standard_Input  => Standard_Input,
          Standard_Output => Standard_Output,
          Standard_Error  => (Standard_Input, Standard_Error))
is -- policy requires that Standard_Output is independent of Standard_Input.
   H : Byte;
   I : Character_Result;
begin
   Aux.Read (Standard_Input, H); -- read from Standard_Input,
   Leak.Leak (H);                -- leak to Standard_Output.
   Aux.Write (Standard_Error, H);-- (needed to make H relevant).
end Main;
```

The `pragma` primitive informs the Ada compiler and the SPARK tools that what follows is SPARK 2014 code. The `with` and `use` primitives specify which modules the present module uses, in our case *Leak* and *Aux*, and a third module containing data types for standard streams. The `with` part of the `Main` procedule specifies which information flows are allowed to occur by running this procedure. The `Global` flow annotation specifies which global state the present procedure affects and how; in our case, the state of standard input, output and error is inspected and modified. A `Depends` clause of the form a `=>` b states that the value of a after running the procedure can depend on the value of b from before the procedure was run. The flow annotation therefore states that information should *not* flow from standard input to standard output. Without the `Aux.Write` line, a component of the Ada compiler which checks for the relevance (i.e. relevant substructural types [50]) of operations rejects the program, on the (false) basis that it reads a value from standard input without using it to produce any result. To show that the analysis overlooks the dependency created by *Leak*, we remove this error message by adding an output to standard error containing the input.

The *Aux* module, `aux.adb`, defines a data type for bytes, and provides a `Read` and a `Write` procedure for standard stream I/O. Its interface specification, `aux.ads`, is as follows.

```
pragma SPARK_Mode (On);                                    -- File: aux.ads
with SPARK.Text_IO; use SPARK.Text_IO;

package Aux is
   type Byte is mod 2**8;
   procedure Write (File: in out File_Type; B : in Byte)
     with Global => null, Depends => (File => (File, B));
   procedure Read (File: in out File_Type; B : out Byte)
     with Global => null, Depends => (File => File, B => File);
end Aux;
```

We introduced this module to hide the defensive programming needed when doing file I/O in SPARK 2014, since SPARK 2014 does not support exceptions.

```
pragma SPARK_Mode (On);                                    -- File: aux.adb
with SPARK.Text_IO; use SPARK.Text_IO;

package body Aux is
   procedure Write (File: in out File_Type; B : in Byte) is
      Output : String (1 .. 1) := "0";
   begin
      Output(1) := Character'Val (B);
      if Is_Writable (File) and then Status (File) = Success then
         Put_Line (File, Output);
      end if;
   end Write;
   procedure Read (File: in out File_Type; B : out Byte) is
      I : Character_Result;
   begin
      if Is_Readable (File) and then not End_Of_File (File) then
         Get (File, I);
         if I.Status = Success
         then B := Byte ( Character'Pos ( I.Item ) );
         else B := 0;
         end if;
      else B := 0;
      end if;
   end Read;
end Aux;
```

The *Leak* module, `leak.adb`, is as described in Section 2. For completeness, the full source of its interface specification, `leak.ads`, follows. Observe that `Main` invokes `Leak` with a variable containing a byte representation of a character read from standard input.

```
pragma SPARK_Mode (On);                                    -- File: leak.ads
with SPARK.Text_IO,Aux; use SPARK.Text_IO,Aux;

package Leak
is
   procedure Leak (H : in out Byte)
     with Global  => (In_Out => Standard_Output),
          Depends => (H => H, Standard_Output => Standard_Output);
end Leak;
```

Explicit and implicit flow. As a first attempt to get `Leak` to leak `H` to standard output, we try to simply write `H` directly to standard out.

```
procedure Leak (H : in out Byte)                              -- procedure in
is                                                            -- File: leak.adb
begin
  H := H;                      -- makes "in" and "out" relevant
  Write (Standard_Output, H); -- explicit flow of H to standard output
end Leak;
```

While we can compile and run this program, the SPARK analysis notices this *explicit flow*, and rejects the program.

```
$ gnatmake main.adb           # compile. ([...] = irrelevant text omitted)
gcc -c main.adb [...]
gnatbind -x main.ali
gnatlink main.ali
$ echo a | ./main 2>/dev/null # run; feed 'a' in on StdIn, ignore StdErr.
a
$ gnatprove -P proj.gpr       # prove; analyze flows & verify contracts.
[...] leak.ads:10:31: warning: missing dependency "Standard_Output => H"
gprbuild: *** compilation phase failed [...]
$
```

Implicit flows, as in the following example, are similarly detected by the SPARK flow analysis.

```
procedure Leak (H : in out Byte)                              -- procedure in
is                                                            -- File: leak.adb
begin
  H := H;                     -- makes "in" and "out" relevant
  if H mod 2 = 1 then         -- implicit flow of H to standard output
     Write (Standard_Output, Character'Pos('1'));
  else
     Write (Standard_Output, Character'Pos('0'));
  end if;
end Leak;
```

Resource exhaustion. In the termination and progress attack, nontermination was a necessary component for circumventing the SPARK flow analysis. We now see whether we can lift this requirement by causing the program to terminate abnormally instead. We replace the infinite loop with a call to a procedure `Blow`, which purpose is to allocate two gigabytes of memory.

```
procedure Leak (H : in out Byte) is          -- procedure in File: leak.adb
   procedure Blow (O : out Byte)
     with Global => null, Depends => (O => null)
   is
      type GByte2 is array (Natural range 0 .. 2147483647) of Byte
        with Default_Component_Value => 255;
      G : GByte2;
   begin O := G(0); end Blow;
   K : Byte := 0;
begin
   H := H;
   while True loop
      Write (Standard_Output, K);
      if K >= H then Blow(H); end if;
      K := K + 1;
   end loop;
end Leak;
```

Main compiles and passes the flow analysis with this implemetation of *Leak*. However, since SPARK allocates all data on the stack, and since the memory

`Blow` needs is allocated on first invocation, the program will terminate abruptly when the character provided on standard input appears on standard output, if the stack is less than two gigabytes (which the default stack size usually is).

```
$ # newlines in standard output replaced with space for brevity.
$ echo "0" | ./main  # stack overflow on outputting 0; program terminates
[...] ( ) * + , - . / 0
raised STORAGE_ERROR : stack overflow or erroneous memory access
$
```

Running `gcc -c -fstack-usage leak.adb ; cat leak.su` reveals the *explicit* allocation of two gigabytes of memory, so a thorough programmer could use this information to either not ship the program, or ensure that the program has enough stack before it is executed. However, this command will not track *implicit* memory allocation, such as the buildup of stack frames through a chain of procedure calls. We replace `Blow` with a procedure which makes an infinite (mutually) recursive call. The resulting program has the same input-output behavior as the previous one, compiles and passes analysis, and `leak.su` states each procedure and function uses less than 100 bytes of stack.

```
procedure Blow (O : out Byte)
  with Global  => null, Depends => (O => null)
is
   function G (B : Byte) return Byte;
   function F (B : Byte) return Byte is begin return G (B) + 1; end F;
   function G (B : Byte) return Byte is begin return F (B) + 1; end G;
begin O := F(0); end Blow;
```

Timing. Finally, we look at how SPARK treats timing. A flow analysis is *timing sensitive* when it tracks whether a value can affect the time an effect occurs. We modify the termination example by replacing the infinite loop by a computation which takes considerable time: selection-sorting 2^{16} bytes.

```
procedure Leak (H : in out Byte)              -- procedure in File: leak.adb
is
   type Bytes is array (Natural range 0 .. 65535) of Byte
     with Default_Component_Value => 255;
   procedure SelectionSort (A : in out Bytes)
     with Global  => null, Depends => (A => A)
   is
      B : Byte;
   begin
      for I in Integer range 0 .. 65535 loop
         for J in Integer range I .. 65535 loop
           if A(J) <= A(I) then B := A(I); A(I) := A(J); A(J) := B; end if;
         end loop;
      end loop;
   end SelectionSort;
   M : Bytes;
begin
   H := H;
   if H mod 2 = 0 then
      for I in Integer range 0 .. 65535 loop
         M(I) := Byte (I mod 256);
      end loop;
      SelectionSort (M);
      if H = M(0) then H := H; end if;
   end if;
   Write (Standard_Output, Character'Pos('!'));
end Leak;
```

Main compiles and passes the flow analysis with this implemetation of *Leak*. Running this program on different values on standard input, however, affects the time it takes for the output to appear on standard output. The flow analysis in SPARK is therefore *timing insenstive*.

```
$ echo 1 | time ./main 2>/dev/null
!
./main  0.00s user 0.00s system 0% cpu 0.001 total
$ echo 0 | time ./main 2>/dev/null
!
./main  10.28s user 0.00s system 100% cpu 10.283 total
$
```

B Semantics

We define the semantics of programs (in terms of expressions) and flow policies in Figure 7.

Expressions. The semantics of expressions is given in terms of a memory $m : \mathbb{X} \times \mathbb{N} \to \mathbb{Z}$. Here, $m(x, n)$ denotes the integer stored in array x at index n. The big-step judgment $m \models e \to n$ states that under m, e evaluates to n. The inference rules defining this judgment are standard, and can be found in Figure 7a.

Programs. The semantics of programs are defined by the small-step judgment $(\boldsymbol{e}, m, p) \xrightarrow{o} (\boldsymbol{e}', m', p')$ in Figure 7b, where $o ::= \bullet \mid !cv$ is an output action. The semantics is the result of tracking assignments to memory at input and assignment blocks while traversing the control flow graph of a program, and consulting the memory at branch points. This can be seen through the following observation: if memories and environments are dropped in the judgment,

$$\frac{}{m \models n \to n} \qquad \frac{m \models e \to n}{m \models x\texttt{[}e\texttt{]} \to m(x,n)}$$

$$\frac{m \models e \to n \qquad m \models e' \to n' \qquad n \odot n' = n''}{m \models e \odot e' \to n''}$$

(a) Expressions

$$\frac{}{(\boldsymbol{e}, m, \texttt{skip; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m, p)}$$

$$\frac{m \models e \to n \qquad m \models e' \to n'}{(\boldsymbol{e}, m, x\texttt{[}e\texttt{]} \texttt{ := } e'\texttt{; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m[(x,n) \mapsto n'], p)}$$

$$\frac{m \models e \to n}{(\boldsymbol{e}, m, c \texttt{ <- } e\texttt{; } p) \xrightarrow{!cn} (\boldsymbol{e}, m, p)}$$

$$\frac{m \models e \to n \qquad s = \boldsymbol{e}(c) \qquad s' = \mathrm{tl}(s) \qquad n' = \mathrm{hd}(s)}{(\boldsymbol{e}, m, c \texttt{ -> } x\texttt{[}e\texttt{]; } p) \xrightarrow{\bullet} (\boldsymbol{e}[c \mapsto s'], m[(x,n) \mapsto n'], p)}$$

$$\frac{m \models e \to n \qquad n \neq 0}{(\boldsymbol{e}, m, \texttt{if } e \texttt{ \{}p_1\texttt{\} \{}p_0\texttt{\}; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m, p_1\texttt{; } p)}$$

$$\frac{m \models e \to n \qquad n = 0}{(\boldsymbol{e}, m, \texttt{if } e \texttt{ \{}p_1\texttt{\} \{}p_0\texttt{\}; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m, p_0\texttt{; } p)}$$

$$\frac{m \models e \to n \qquad n \neq 0}{(\boldsymbol{e}, m, \texttt{while } e \texttt{ \{}p_1\texttt{\}; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m, p_1\texttt{; while } e \texttt{ \{}p_1\texttt{\}; } p)}$$

$$\frac{m \models e \to n \qquad n = 0}{(\boldsymbol{e}, m, \texttt{while } e \texttt{ \{}p_1\texttt{\}; } p) \xrightarrow{\bullet} (\boldsymbol{e}, m, p)}$$

(b) Programs

$$\begin{aligned}
[\![\texttt{null}]\!] &= (\emptyset, \emptyset) \\
[\![c \texttt{ => } c'\texttt{; } f]\!] &= (\kappa \cup \{(c,c')\}, \pi), \text{ where } [\![f]\!] = (\kappa, \pi) \\
[\![c \texttt{ -> } c'\texttt{; } f]\!] &= (\kappa, \pi \cup \{(c,c')\}), \text{ where } [\![f]\!] = (\kappa, \pi)
\end{aligned}$$

(c) Policies

Fig. 7: Semantics

all premises replaced with "true", and $\xrightarrow{o}$ replaced with $\to$ for all o, the resulting rules are *exacly* those defining the CFG in Figure 2 (right). Thus, if $(\boldsymbol{e}, m, p) \xrightarrow{o} (\boldsymbol{e}', m', p')$, then $p \to p'$ is an edge in the CFG. Following the abbreviated notation for CFG nodes introduced in Section 3, we also get a stepping

relation on blocks, $(\boldsymbol{e}, m, b) \xrightarrow{o} (\boldsymbol{e}', m', b')$. We use this relation extensively in our proofs. Inputs on channels are provided by a stream environment, $\boldsymbol{e} : \mathbb{C} \to \mathbb{Z}^\omega$. Intuitively, $\boldsymbol{e}(c) = v_1 v_2 \ldots$ is the sequence of inputs on channel c that have not been consumed by the program yet. For an infinite sequence $s = v_1 v_2 v_3 \ldots$, we define $\mathrm{hd}(s) = v_1$ (head) and $\mathrm{tl}(s) = v_2 v_3 \ldots$ (tail).

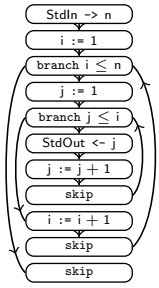
Flow policies A flow policy specification f defines a flow policy $(\pi, \kappa) = [\![f]\!]$ denotationally in Figure 7c. The resulting κ and π are *relations* on channels. We then interpret these as functions as follows: $\kappa(c) = \{c' \mid (c, c') \in \kappa\}$, and $\pi(c) = \{c' \mid (c, c') \in \pi\}$.

C Examples

Example 1. An output of `0` to standard output can be modeled in our formalism as the output action !StdOut0.

Example 2. The program on the right reads an integer `n` from standard input and, for each $1 \leq \mathsf{i} \leq \mathsf{n}$, outputs $1, \ldots, \mathsf{i}$ to standard output. The graph next to it is the CFG of this program.

```
StdIn -> n;
i := 1;
while i ≤ n {
   j := 1;
   while j ≤ i {
      StdOut <- j;
      j := j + 1;
      skip
   };
   i := i + 1;
   skip
};
skip
```



Example 3. Let p = `StdIn -> n;` p' be the program in Example 2, and let $\boldsymbol{e}(\mathsf{StdIn}) = 4.s$ for some $\boldsymbol{e}$ and s. Then

$$(\boldsymbol{e}, m_0, p) \xrightarrow{\bullet} (\boldsymbol{e}[\mathsf{StdIn} \mapsto \mathrm{s}], m_0[(\mathsf{n}, \mathsf{d}) \mapsto 4], p')$$

by the fourth rule of the semantics of p. This transition corresponds to traversing the edge in the CFG in Example 2 from the root node to the node labeled `i := 1` (since $p' =$ `i := 1;` p'' for some p'').

Example 4. A flow policy stating that (only) the presence of output on StdErr (standard error) is allowed to depend on input on StdIn (standard input) can be written as `StdIn -> StdErr; null`.

Example 5. Let p be a program that behaves like the progress leak in Section 2, $f =$ `StdErr => StdIn; null`, $\boldsymbol{e}_1 = \boldsymbol{e}[\mathsf{StdIn} \mapsto s_1]$ and $\boldsymbol{e}_2 = \boldsymbol{e}[\mathsf{StdIn} \mapsto s_0]$, where $\boldsymbol{e}$ is arbitrary, $s_1 = 1.s_1$ and $s_0 = 0.s_0$. Consider $\omega_\pi = \omega_\kappa = \{\mathsf{StdOut}\}$. Since ω does not observe StdErr, StdIn is unobservable to ω. So for this ω, $\boldsymbol{e}_1 \sim_\omega \boldsymbol{e}_2$. However, $(\boldsymbol{e}_1, m_0, p)$ and $(\boldsymbol{e}_2, m_0, p)$ cannot observably match all behaviors; while $(\boldsymbol{e}_1, m_0, p) \xrightarrow{\bar{o}_1'}$ for some $\bar{o}_1'$ with

$$\bar{o}_1' \restriction_\omega = \bar{o_1} = \cdots.!\mathsf{StdOut}.\,.!\mathsf{StdOut}/.!\mathsf{StdOut0}.!\mathsf{StdOut1},$$

$(\boldsymbol{e}_2, m_0, p)$ only matches $\bar{o}_1$ up to 0, i.e. produces

$$\bar{o_2}'\restriction_\omega = \bar{o_2} = \cdots.!\mathsf{StdOut}..!\mathsf{StdOut}/.!\mathsf{StdOut0}$$

at most; any other $\bar{o}$ produced by $(\boldsymbol{e}_2, m_0, p)$ strictly observably prefixes $\bar{o}_2$ which strictly observably prefixes $\bar{o}_1$, and therefore cannot be observably equivalent with $\bar{o}_1$. So, for all $\bar{o}_2'$ for which $(\boldsymbol{e}_2, m_0, p) \xrightarrow{\bar{o}_2'}$, $\bar{o}_1' \not\sim_\omega \bar{o}_2'$. Thus, p is not PSNI.

D Proof of Theorem 1

In this section, we prove Theorem 1. We start with several auxiliary definitions and lemmas.

Definition 10 (Slice, denoted S). *A slice of a CFG G is a subgraph S of G satisfying the following condition: If $b \in S$ and $(b', b) \in dep_G$, then $b' \in S$.*

Note that, by definition, $\mathrm{BS}_G(b)$ is a slice for any $b \in G$. Amtoft [1] proves the following lemma.

Lemma 1 (Unique entry point [1, Lemma 5]). *For any slice S of G and $b \notin S$, there is at most one node $b' \in S$ such that (1) $b \to^* b' \in G$, and (2) all nodes other than b' on any path between b and b' in b are not in S.*

Lemma 1 implies that starting from any node b (inside or outside S), there is at most one "entry point" into S (when $b \in S$, this entry point is b itself). We write $\mathrm{ep}_S(b) = b'$ when this unique entry point is b' and write $\mathrm{ep}_S(b) = \bot$ when S is not reachable from b.

Lemma 2. *If $b \notin S$ and $b \to b'$, then $ep_S(b) = ep_S(b')$.*

Proof. We consider three cases. If $\mathrm{ep}_S(b) = \bot$, then there is no path from b to any node in S and, hence, there cannot be a path from b' to any node in S. Therefore, $\mathrm{ep}_S(b') = \bot = \mathrm{ep}_S(b)$.

If $\mathrm{ep}_S(b) = b''$ and $\mathrm{ep}_S(b') \neq \bot$, then by Lemma 1, we must have $\mathrm{ep}_S(b') = b''$.

If $\mathrm{ep}_S(b) = b''$ and $\mathrm{ep}_S(b') = \bot$, then there is a path from b to b'' but not through b'. The latter implies that b'' does not post-dominate b. It follows that b'' is control-dependent on b. Since S is a slice and $b'' \in S$, we must also have $b \in S$, which is a contradiction. Hence, this case is impossible. □

We now define equivalence of configurations of the form $(\boldsymbol{e}, m, b)$, with respect to a slice S.

Definition 11 (State equivalence). *Given a slice S, $(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_2, m_2, b_2)$ if the following hold:*

1. *$ep_S(b_1) = ep_S(b_2)$. In particular, this implies that the sets of nodes of S reachable from b_1 and b_2 are equal.*

2. *If a node in S reachable from b_1 (or b_2) reads array x, then for all n, $m_1(x,n) = m_2(x,n)$.*
3. *If a node in S reachable from b_1 (or b_2) reads channel c, then $\boldsymbol{e}_1(c) = \boldsymbol{e}_2(c)$.*

Lemma 3 (Confinement). *If $(\boldsymbol{e}_1, m_1, b_1) \to (\boldsymbol{e}_2, m_2, b_2)$ and $b_1 \notin S$, then $(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_2, m_2, b_2)$.*

Proof. By analysis of the rules that apply to $\to$, we show that (1) – (3) of Definition 11 hold. Briefly, (1) holds from Lemma 2.

(2) can be violated only when some array x whose read in S is reachable from b_1 is assigned in b_1. Suppose such an array is read at node b' in S. Now, b' is data dependent on b_1, so $b_1 \in S$ by the definition of slice. This is a contradiction as $b_1 \notin S$ by assumption.

(3) can be violated only when some channel c whose read in S is reachable from b_1 is read in b_1. Suppose c is read at node b' in S. Now, b' is data dependent on b_1, so $b_1 \in S$ by the definition of slice. This is a contradiction as $b_1 \notin S$ by assumption. □

Notation. We write $(\boldsymbol{e}_1, m_1, b_1) \xrightarrow{o}_S (\boldsymbol{e}_2, m_2, b_2)$ when $(\boldsymbol{e}_1, m_1, b_1) \xrightarrow{o} (\boldsymbol{e}_2, m_2, b_2)$ and $b_1 \in S$. We write $(\boldsymbol{e}_1, m_1, b_1) \to_{\not S} (\boldsymbol{e}_2, m_2, b_2)$ when $(\boldsymbol{e}_1, m_1, b_1) \xrightarrow{\_} (\boldsymbol{e}_2, m_2, b_2)$ and $b_1 \notin S$. Here, $\_$ denotes $\bullet$ or $!cv$. Finally, we define $\Rightarrow^o_S = \to^*_{\not S} \circ \xrightarrow{o}_S$.

Definition 12 (Progress-complete CFG). *A CFG G is called progress complete, written $pc(G)$, if for every $b \in G$, and every $\boldsymbol{e}, m$, if b appears infinitely often on the reduction sequence starting from $(\boldsymbol{e}, m, \mathsf{START})$, then there is an edge from b to END.*

Note that if T is a sound termination oracle, then it must be the case that $\mathrm{pc}(\mathrm{ps}_T(G))$.

Lemma 4 (Simulation). *Suppose $pc(G)$ and S is a slice of G. If*

$$(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_2, m_2, b_2) \text{ and } (\boldsymbol{e}_1, m_1, b_1) \Rightarrow^o_S (\boldsymbol{e}'_1, m'_1, b'_1),$$

then there exist $(\boldsymbol{e}'_2, m'_2, b'_2)$ such that

$$(\boldsymbol{e}_2, m_2, b_2) \Rightarrow^o_S (\boldsymbol{e}'_2, m'_2, b'_2) \text{ and } (\boldsymbol{e}'_1, m'_1, b'_1) \sim_S (\boldsymbol{e}'_2, m'_2, b'_2).$$

Proof. From $(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_2, m_2, b_2)$, we know that $\mathrm{ep}_S(b_1) = \mathrm{ep}_S(b_2)$. If $\mathrm{ep}_S(b_1) = \mathrm{ep}_S(b_2) = \bot$, then $(\boldsymbol{e}_1, m_1, b_1) \Rightarrow^o_S (\boldsymbol{e}'_1, m'_1, b'_1)$ cannot exist. Hence, there is some b'' such that $\mathrm{ep}_S(b_1) = \mathrm{ep}_S(b_2) = b''$. We consider two cases.

Case $b_1, b_2 \in S$. Then, $b_1 = \mathrm{ep}_S(b_1) = b'' = \mathrm{ep}_S(b_2) = b_2$. By conditions (2) and (3) of Definition 11, all arrays (channels) read by $b_1 (= b_2)$ are equal in m_1/m_2 ($\boldsymbol{e}_1/\boldsymbol{e}_2$). It immediately follows that we can choose $b'_2 = b'_1$ and $(\boldsymbol{e}'_2, m'_2, b'_2)$ as the (unique) successor state of $(\boldsymbol{e}_2, m_2, b_2)$ to satisfy the theorem.

Case $b_1 \notin S$ or $b_2 \notin S$. Then, $(\boldsymbol{e}_1, m_1, b_1) \to^*_{\not S} (\boldsymbol{e}''_1, m''_1, b'') \xrightarrow{o}_S (\boldsymbol{e}'_1, m'_1, b'_1)$. Using Lemma 3, we get:

1. $(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_1'', m_1'', b'')$

Now we claim that on any path from b_2 to $\mathrm{ep}_S(b_2) = b''$, there is no node of an infinite loop other than possibly b'' itself. Why? If there were a node b of an infinite loop on a path from b_2 to b'', then due to the property $\mathrm{pc}(G)$, there would be an edge from b to END, which would make b'' control dependent on b. Hence, by the definition of slice, b would be in S. This would force $b = b''$ as no other node on a path from b_2 to $b'' = \mathrm{ep}_S(b_2)$ can be in S.

Consequently, assuming that the language is type safe, there must be a (finite) reduction sequence $(\boldsymbol{e}_2, m_2, b_2) \to_{\mathscr{G}}^* (\boldsymbol{e}_2'', m_2'', b'')$. Using Lemma 3 we get:

2. $(\boldsymbol{e}_2, m_2, b_2) \sim_S (\boldsymbol{e}_2'', m_2'', b'')$

(1), (2) and the given condition $(\boldsymbol{e}_1, m_1, b_1) \sim_S (\boldsymbol{e}_2, m_2, b_2)$ together imply that $(\boldsymbol{e}_1'', m_1'', b'') \sim_S (\boldsymbol{e}_2'', m_2'', b'')$. Now we reason as in the previous case to argue for existence of the required $(\boldsymbol{e}_2', m_2', b_2')$ observing that $(\boldsymbol{e}_1'', m_1'', b'') \xrightarrow{o}_S (\boldsymbol{e}_1', m_1', b_1')$. □

In the sequel, we assume a fixed policy (κ, π). We define a predicate $\mathsf{check}'(G)$ that is similar to the predicate $\mathsf{check}_T(G)$, but defined on a graph G to which additional edges have already been added.

Definition 13 (Predicate $\mathsf{check}'(G)$). *We say that $\mathsf{check}'(G)$ when the following conditions hold for any node b of the form c `<-` e in G:*

1. *If c' `->` x `[`e'`]` $\in BS_G(b)$ then $(c, c') \in \kappa$.*
2. *If c' `->` x `[`e'`]` $\in BS_{\hat{G}}(\hat{b})$ then $(c, c') \in \pi$, where $\hat{G}$ is obtained by replacing b with $\hat{b} = c$ `<-` `d` in G.*

Lemma 5 (Connection between check_T and check'). *If $\mathsf{check}_T(p)$, then $\mathsf{check}'(G')$ where G is the CFG of p and $G' = ps_T(G)$.*

Proof. Immediate from Definitions 9 and 13. □

For $\bar{o} = o_1 \ldots o_n$, we define $\xrightarrow{\bar{o}} = (\xrightarrow{\bullet})^* \xrightarrow{o_1} (\xrightarrow{\bullet})^* \ldots (\xrightarrow{\bullet})^* \xrightarrow{o_n} (\xrightarrow{\bullet})^*$. Next, we define an auxiliary graph $G{\restriction_\omega}$, which syntactically erases all output content that the adversary ω cannot see.

Definition 14 (Reduced graph, $G{\restriction_\omega}$). *Define the graph $G{\restriction_\omega}$ as follows: Replace every node of the form c `<-` e in G with c `<-` `d` if $c \notin \omega_\kappa$.*

Lemma 6. *The following hold:*

1. *If $C \xrightarrow{\bar{o}}$ in G, then $C \xrightarrow{\bar{o}'}$ in $G{\restriction_\omega}$ with $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$.*
2. *If $C \xrightarrow{\bar{o}}$ in $G{\restriction_\omega}$, then $C \xrightarrow{\bar{o}'}$ in G with $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$.*

Proof. Immediate, observing that $G{\restriction_\omega}$ is obtained from G by erasing outputs to channels on which the adversary ω cannot observe content. Formally, the proof follows by induction on the given sequences $\xrightarrow{\bar{o}}$ and $\xrightarrow{\bar{o}'}$, respectively. The environments and memories on the two sides of the simulation remain in perfect sync because replacing c `<-` e with c `<-` `d` does not impact the program's execution (our environments are streams, not adaptive functions). □

We define a third check_T-like predicate, which uses $G{\restriction_\omega}$ instead of G.

Definition 15 (Predicate $\mathsf{check}''(G,\omega)$). *Let $G' = G{\restriction_\omega}$. We say that $\mathsf{check}''(G,\omega)$ when the following conditions hold for any node b of the form c `<-` e in G' and any node of the form c' `->` x `[`e'`]` in $BS_{G'}(b)$:*

1. *If $c \in \omega_\kappa$, then $c' \in \kappa(c)$.*
2. *If $c \in \omega_\pi$, but $c \notin \omega_\kappa$, then $c' \in \pi(c)$.*

Lemma 7 (Relation between check' and check''). $\mathsf{check}'(G)$ *implies* $\mathsf{check}''(G,\omega)$ *for any* ω.

Proof. Assume $\mathsf{check}'(G)$. Let $G' = G{\restriction_\omega}$. To show $\mathsf{check}''(G,\omega)$, pick any node b of form c `<-` e in G' and any node b' of form c' `->` x`[`e'`]` in $\mathrm{BS}_{G'}(b)$. We have to show the following: (1) If $c \in \omega_\kappa$, then $c' \in \kappa(c)$, and (2) If $c \in \omega_\pi$, but $c \notin \omega_\kappa$, then $c' \in \pi(c)$.

Proof of (1): Assume $c \in \omega_\kappa$. From $b' \in \mathrm{BS}_{G'}(b)$, it follows that $(b', b) \in \mathrm{dep}_{G'}$. Since b occurs as-is in G as well (because $c \in \omega_\kappa$), we must have $(b', b) \in \mathrm{dep}_G$. Hence, $b' \in \mathrm{BS}_G(b)$ and, by $\mathsf{check}'(G)$, we get $c' \in \kappa(c)$, as required.

Proof of (2): Assume $c \in \omega_\pi$ and $c \notin \omega_\kappa$. Since $c \notin \omega_\kappa$, it follows from $G' = G{\restriction_\omega}$ that $e = \mathsf{d}$. Let b'' be the node corresponding to b in G. Note that b'' must have form c `<-` e'' for some e''. Hence, the $\hat{b}$ referred to in clause (2) of Definition 13 is exactly the b here. Since G' is an erasure of G, $b' \in \mathrm{BS}_{G'}(b)$ implies $b' \in \mathrm{BS}_G(b)$. From $\mathsf{check}'(G)$, we immediately get $c' \in \pi(c)$. □

Definition 16. *For an attacker ω and a graph G, we define the adversarial slice of $G{\restriction_\omega}$ (not G) as the set of all nodes of $G{\restriction_\omega}$ (not G) that may influence an ω-visible output. Formally, $aS_\omega(G) = \{b' \mid b = (c \texttt{ <- } e) \in G{\restriction_\omega} \wedge b' \in BS_{G{\restriction_\omega}}(b) \wedge c \in \omega_\pi\}$.*

Note that $\mathrm{aS}_\omega(G)$ is a union of backward slices, each of which is closed under $(\mathrm{dep}_{G{\restriction_\omega}})^{-1}$. Therefore, $\mathrm{aS}_\omega(G)$ is also closed under $(\mathrm{dep}_{G{\restriction_\omega}})^{-1}$. Consequently, $\mathrm{aS}_\omega(G)$ is a slice in the sense of Definition 10.

Lemma 8 (Initial equivalence). *Suppose that $\mathsf{check}''(G,\omega)$, $\boldsymbol{e}_1 \sim_\omega \boldsymbol{e}_2$ and $S_A = aS_\omega(G)$. Then for any m and b, $(\boldsymbol{e}_1, m, b) \sim_{S_A} (\boldsymbol{e}_2, m, b)$.*

Proof. It suffices to prove condition (3) of Definition 11 since conditions (1) and (2) are trivial here. So pick any c' that is used in a node of S_A reachable from b in $G{\restriction_\omega}$. It suffices to prove that $\boldsymbol{e}_1(c') = \boldsymbol{e}_2(c')$. By the definition of $S_A = \mathrm{aS}_\omega(G)$, there must be some node b' of the form c `<-` e in $G{\restriction_\omega}$ such that c' is used in $\mathrm{BS}_{G{\restriction_\omega}}(b')$ and $c \in \omega_\pi$. Since c' is used in $\mathrm{BS}_{G{\restriction_\omega}}(b')$, there is some node of form c' `->` x`[`e'`]` in $\mathrm{BS}_{G{\restriction_\omega}}(b')$. Now, two cases arise:

Case $c \in \omega_\kappa$. By $\mathsf{check}''(G,\omega)$, $c' \in \kappa(c)$. By definition of $\boldsymbol{e}_1 \sim_\omega \boldsymbol{e}_2$, it follows immediately that $\boldsymbol{e}_1(c') = \boldsymbol{e}_2(c')$, as required.

Case $c \notin \omega_\kappa$. Because $c \in \omega_\pi$, from $\mathsf{check}''(G,\omega)$ we get $c' \in \pi(c)$. By definition of $\boldsymbol{e}_1 \sim_\omega \boldsymbol{e}_2$, it follows immediately that $\boldsymbol{e}_1(c') = \boldsymbol{e}_2(c')$, as required. □

Theorem 2. *Let $S_A = aS_\omega(G)$ and pc(G). Suppose $(\boldsymbol{e}, m, b) \sim_{S_A} (\boldsymbol{e}', m', b')$. If $(\boldsymbol{e}, m, b) \xrightarrow{\bar{o}}$ in $G{\restriction_\omega}$, then there exists an $\bar{o}'$ such that $(\boldsymbol{e}', m', b') \xrightarrow{\bar{o}'}$ in $G{\restriction_\omega}$ and $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$.*

Proof. First note that pc(G) implies pc$(G{\restriction_\omega})$. This means that Lemma 4 is applicable to $G{\restriction_\omega}$ and S_A. We are given $(\boldsymbol{e}, m, b) \xrightarrow{\bar{o}}$ in $G{\restriction_\omega}$. Write this sequence as $(\boldsymbol{e}, m, b) = s_0 \Rightarrow^{o_1}_{S_A} s_1 \ldots \Rightarrow^{o_n}_{S_A} s_n$. By Lemma 4, there is a reduction sequence $(\boldsymbol{e}', m', b') = s'_0 \Rightarrow^{o_1}_{S_A} s'_1 \ldots \Rightarrow^{o_n}_{S_A} s'_n$ such that for each i, $s_i \sim_{S_A} s'_i$. We choose $\bar{o}'$ to be all outputs in this reduction sequence. Note that the two reduction sequences may have outputs other than in $o_1 \ldots o_n$. For instance, $s'_0 \Rightarrow^{o_1}_{S_A} s'_1$ may have outputs before the last output o_1. However, by construction of S_A, if any node b outputs on a channel c such that $c \in \omega_\kappa \cup \omega_\pi$ (which implies $c \in \omega_\pi$), then $b \in S_A$. So, any output in $\bar{o}{\restriction_\omega}$ or in $\bar{o}'{\restriction_\omega}$ must be in $o_1 \ldots o_n$. It follows immediately that $\bar{o}{\restriction_\omega} = (o_1 \ldots o_n){\restriction_\omega} = \bar{o}'{\restriction_\omega}$. □

Corollary 1. *Suppose* $\mathsf{check}'(G)$*, pc(G) and $\boldsymbol{e} \sim_\omega \boldsymbol{e}'$. Let m be any memory. If $(\boldsymbol{e}, m, \mathsf{START}) \xrightarrow{\bar{o}}$, then there exists $\bar{o}'$ such that $(\boldsymbol{e}', m, \mathsf{START}) \xrightarrow{\bar{o}'}$ and $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$.*

Proof. Let $S_A = \mathrm{aS}_\omega(G)$. From $\mathsf{check}'(G)$ and Lemma 7, we get $\mathsf{check}''(G, \omega)$. From Lemma 8 we get:

1. $(\boldsymbol{e}, m, \mathsf{START}) \sim_{S_A} (\boldsymbol{e}', m, \mathsf{START})$.

From $(\boldsymbol{e}, m, \mathsf{START}) \xrightarrow{\bar{o}}$ and Lemma 6(1), there is a $\bar{o}_1$ such that:

2. $(\boldsymbol{e}, m, \mathsf{START}) \xrightarrow{\bar{o}_1}$ in $G{\restriction_\omega}$
3. $\bar{o}{\restriction_\omega} = \bar{o}_1{\restriction_\omega}$

By Theorem 2 applied to (1) and (2), we get a sequence $\bar{o}_2$ such that:

4. $(\boldsymbol{e}', m, \mathsf{START}) \xrightarrow{\bar{o}_2}$ in $G{\restriction_\omega}$
5. $\bar{o}_1{\restriction_\omega} = \bar{o}_2{\restriction_\omega}$

From Lemma 6(2) applied to (4), there is a $\bar{o}'$ such that:

6. $(\boldsymbol{e}', m, \mathsf{START}) \xrightarrow{\bar{o}'}$ in G
7. $\bar{o}_2{\restriction_\omega} = \bar{o}'{\restriction_\omega}$

From (3), (5), (7) we get

8. $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$

We are done by (6) and (8). □

Theorem 3 (Soundness of enforcement, Theorem 1). *If T is a sound termination oracle and* $\mathsf{check}_T(p)$*, then p satisfies* PSNI.

Proof. Let G be the CFG of p and $G' = \mathrm{ps}_T(G)$. From Lemma 5, we get $\mathsf{check}'(G')$. Further from the soundness of T, we get pc(G'). Finally, note that for G', $\mathsf{START} = p$ by definition.

Now, we unfold the definition of PSNI. We pick $\boldsymbol{e}, \boldsymbol{e}'$ such that $\boldsymbol{e} \sim_\omega \boldsymbol{e}'$ and $\bar{o}$ such that $(\boldsymbol{e}, m_0, p) \xrightarrow{\bar{o}}$, i.e., $(\boldsymbol{e}, m_0, \mathsf{START}) \xrightarrow{\bar{o}}$. By Corollary 1 applied to G', we get $\bar{o}'$ such that $(\boldsymbol{e}, m_0, \mathsf{START} = p) \xrightarrow{\bar{o}'}$ and $\bar{o}{\restriction_\omega} = \bar{o}'{\restriction_\omega}$. This is exactly what the definition of PSNI requires us to show. □